"Refactoring" Refactoring

Leslie J. Waguespack[1], Jeffry S. Babb[2] & David J. Yates[1]
[1]Bentley University *[2]West Texas A&M University*
Waltham, Massachusetts, USA *Canyon, Texas, USA*
lwaguespack@bentley.edu *jbabb@wtamu.edu* *dyates@bentley.edu*

Abstract

Code refactoring's primary impetus is to control technical debt, a metaphor for the cost in software development due to the extraneous human effort needed to resolve confusing, obfuscatory, or hastily-crafted program code. While these issues are often described as causing "bad smells," not all bad smells emanate from the code itself. Some (often the most pungent and costly) originate in the formation, or expressions, of the antecedent intensions the software proposes to satisfy. Paying down such technical debt requires more than grammatical manipulations of the code. Rather, refactoring in this case must attend to a more inclusive perspective; particularly how stakeholders perceive the artifact; and their conception of quality – their appreciative system. First, this paper explores refactoring as an evolutionary design activity. Second, we generalize, or "refactor," the concept of code refactoring, beyond changes to code structure, to improving design quality by incorporating the stakeholders' experience of the artifact as it relates to their intensions. Third, we integrate this refactored refactoring as the organizing principle of design as a reflective practice. The objective is to improve the clarity, understandability, maintainability, and extensibility manifest in the stakeholder intensions, in the artifact, and in their interrelationship!

1. Introduction

Code refactoring is the act of modifying the grammatical structure of source code while retaining the code's existing behavior. According to Fowler et al.:

> "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written." [14]

Code refactoring reveals the quintessence of evolutionary software development [15]. It is the incremental application of design actions in pursuit of quality guided by the regard and diagnosis of undesirable design affects (aka "bad smells" – so named to connote a foul situation in need of attention) [8, 37]. In spite of the importance of refactoring as a tool to iteratively design and develop object-oriented software, little has been done to generalize refactoring beyond work that is specific to application frameworks [28], models [19, 36], and product lines [3].

This work proposes using a theory of design, *Thriving Systems Theory (TST)* [40], to generalize code refactoring to *design refactoring*. We examine 80 refactorings presented by Fowler, Beck, Kerievsky, and others [14, 20] and show that they map primarily to six of the 15 design choice properties promulgated by TST. Thus, that the remaining nine choice properties were found not to align with refactoring both informs the nature of our inquiry and forms the basis of our generalization. As a theory of design, Thriving Systems Theory recognizes the role of reflective practice [34] as a key ingredient in the process of iterative product design and development [22]. Even though this is an important part of methodologies like Agile software development [7, 12], a theoretical understanding will strengthen our ability to assess the evolving quality in artifact design.

As we generalize code refactoring to design refactoring, we leverage Lee and Baskerville [23] to generalize refactoring from a limited conceptualization, rooted in technical rationality [34], to a broader design theory (TST) that can be used:

- At different levels of abstraction using the same principles [25, 33, 35];
- To generate domain specific ontologies (e.g., for object-oriented programming) using a common taxonomy [9, 38, 41]; and
- In iterative software delivery to harmonize changing stakeholder intensions & requirements, design actions, and reflective practice [12, 22, 39].

In reflective practice, this generalization applies to refactorings that are in the abstract or instance domains [24, 16, 19, 33] and are holistic or prescriptive [14, 20,

URI: http://hdl.handle.net/10125/41731
ISBN: 978-0-9981331-0-2

37]. We illustrate how this generalization might be useful to inform reflective practice when refactoring in iterative software delivery, e.g., based on Agile development [6, 27, 32] and DevOps [21]. Within the design, development and delivery cycle, the reflective practice considers structural (e.g., modularity and cohesion [5, 44]), behavioral (e.g., correctness and reliability [18, 29]), and aesthetic properties of the software.

2. Aspects of design quality

The expressed goal of code refactoring is to improve the software's design quality – to increase its "value." Understanding the relationship between refactoring and design quality requires an interpretive framework. The design choice properties presented in Thriving Systems Theory (TST) are presented as a vocabulary to further this discussion. [40] (TST's choice properties derive from Christopher Alexander's *Nature of Order*. [2, p. 80])

TST specifies fifteen properties of design choices that differentiate aspects of quality in an artifact. (See Table 1.) Each property reflects design quality, depicted and reinforced by means of a generic design action that when applied effectively, intensifies that property to enhance the design quality of the artifact as a whole. To apply TST as a quality lens to the design choices of a particular artifact the generic design actions must be transliterated as actions in the artifact's specific formative paradigm, the dimensions of its existence (i.e. its context, purpose, behavior, medium of description, means of construction, etc.).

Computing professionals readily recognize the first six properties in Table 1 as desirable structural qualities in documentation, source code, process models, data models, or any overall organization. They are characteristics amenable to being quantified [11]. The remaining nine properties basically defy quantification because they mostly evoke personal, emotional, or psychological reactions mediated by culture, education, and experience that form a person's *world-view*.

In Figure 1, TST's fifteen choice properties appear on the circle's circumference. Again, six of these properties articulate structural qualities of design while the other nine articulate quality aspects of a more aesthetic nature that involve subjective assessment. The confluence of the six properties of design structure appears in the convergence of the pair-wise combination of their property affects – shaded in salmon. The six

	Choice Property	Design Action	Generic Action Definition	Refactoring Involvement
Structural Properties	Modularization	Modularize	employing or involving a module or modules as the basis of design or construction	31
	Cohesion	Factor	express as a product of factors	23
	Encapsulation	Encapsulate	enclose the essential features of something succinctly by a protective coating or membrane	36
	Composition of Function	Assemble	fit together the separate component parts of (a machine or other object)	11
	Stepwise Refinement	Elaborate	develop or present (a theory, policy, or system) in detail	3
	Scale	Focus	(of a person or their eyes) adapt to the prevailing level of light [abstraction] and become able to see clearly	0
Aesthetic Properties	Identity	Identify	establish or indicate who or what (someone or something) is	4
	Patterns	Pattern	give a regular or intelligible form to	0
	Programmability	Generalize	make or become more widely or generally applicable	0
	User Friendliness	Accommodate	fit in with the wishes or needs of	0
	Reliability	Normalize	make something more normal, which typically means conforming to some regularity or rule	0
	Correctness	Align	put (things) into correct or appropriate relative positions	0
	Transparency	Expose	reveal the presence of (a quality or feeling)	0
	Extensibility	Extend	render something capable of expansion in scope, effect, or meaning	0
	Elegance	Coordinate	bring the different elements of (a complex activity or organization) into a relationship that is efficient or harmonious	0

Table 1 – Thriving Systems Theory and Refactoring Instances

properties cluster to articulate progressively more complex, structural qualities described in greater detail in [40]. The remaining nine properties likewise cluster to articulate increasingly complex aesthetic qualities more readily interpreted through analogy – shaded in green. In progressive pair-wise composition the clusters *frame* and *name* the quality affect that the stakeholders experience of the artifact in comprehending it, in using it, and in adapting or modifying it – where the culminating fusion of all the properties is *thriving*.

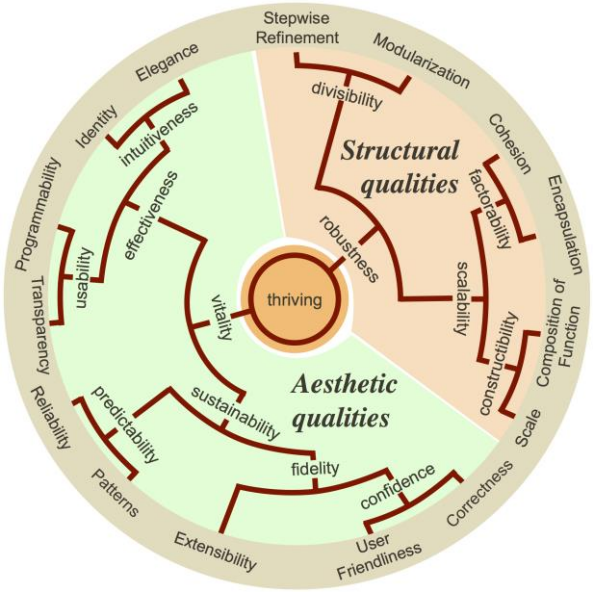

Figure 1 – Choice Properties Conflating as Design Quality Clusters

Observers apprehend the affects of an artifact's choice properties in confluence as a composite sense of design quality. Each observer perceives the import of the properties personally – through their own "eye of the beholder." While every design exhibits all of these fifteen properties, each observer may interpret each property's significance somewhere in the range of barely perceptible through pronounced conspicuousness. It is the skilled designer who discerns in the overall quality the impact of individual choice properties while a novice observer may sense quality, but only as a blur of property affects.

3. "Bad smells" as design quality deficit

The enumeration of refactorings provided by others [14, 20] is their effort to specify what they understand of their tacit knowledge of "bad smells." They propose to both explain and communicate their tacit knowledge of refactoring to others who aspire to learn the skill. Based upon their summary of each of 80 situations needing refactoring, and the consequent refactoring actions they prescribe, we are able to identify Fowler et al.'s and Kerievsky's refactorings that attend to specific TST choice properties [14, 20]. Table 1 tallies the number of refactorings that associate with specific choice properties. Albeit that every choice property plays some role in any design choice, most of their refactorings adhere primarily to an individual property, a few to two properties and one registers three properties.

All but four instances of these refactorings adhere to choice properties that characterize "structural" code manipulation, grammatical in nature, to resituate or reorient syntactic elements of troublesome code. Remediation is prescribed as primarily grammatical or structural often amenable to formulaic manipulations that may be automated in code editors or IDE's [13, 25]. The four non-structural exceptions are instances where refactoring actions apply "naming" or "renaming" of code elements to clarify or signify the programmer's motivation; these refactorings incorporate knowledge of the purpose of the code that cannot be gleaned from the source code alone in either the activity of recognition or remediation. Rather, they infer what the code is "meant" to achieve.

Our exercise of identifying refactorings adhering to respective choice properties points to technical debt [8, 37] that reflects design choices deficient in some corresponding aspect of design quality. The prescribed refactoring applies design actions to remediate the deficiency. The alignment of the specific design actions that Fowler, Beck, Kerievsky and others prescribe in each of their refactorings with design quality choice properties indicates that the detection, diagnosis, as well as the treatment of bad smells derive from their tacit theory of a specific choice property's role in design quality [37]. This offers a clear example of tacit knowledge [30] acquired by dealing with and resolving myriad incidents of "breakdown," failures in program code to effectively and efficiently express the stakeholders' intensions (stakeholders: users, clients, analysts, programmers) [43]. Left unresolved, these code breakdowns incur extraneous effort each time the code is attended to; thus amounting to technical debt, and perhaps "design debt" [8, p. 50, 20, p. 15], that accumulates as cost associated with code maintenance and inevitably erodes the code's value. It is further noteworthy that, in code refactoring, the prevailing tacit theory of design quality is almost entirely confined to the structural properties of program code.

This finding that refactorings in the literature are nearly exclusively structural in nature proceeds from the prescription that code refactoring must retain the code's behavior unchanged [14, 20, 37]. And thus, absent any specific foreknowledge of the authoring programmer's intended purpose of the code (or what Peter Naur calls

"the theory of the program" [26]), refactoring must treat the code as fully self-evident *as-is*. Code refactoring in the philosophical terms of objectivist hermeneutics, suggests "all the meaning of the text resides in the text" [43, p. 27]. Code refactoring (particularly automated refactoring) relies upon this self-evident "meaning" of the source code. In this sense, code refactoring is useful only when quality deficits are completely self-contained in the code itself.

4. Unpacking code refactoring

Code refactoring applies only manipulations that result in a semantically equivalent transformation – the "meaning" of the code (evidenced by its behavior) does not change. In effect, programmers refactor grammatically valid code to render a different grammatically valid code but, without any semantic change. The possible transformations that achieve this semantic isomorphism are defined and confined by the code's language paradigm that frames the expressible structure and behavior. The language paradigm (in this case object-orientation) is de facto both a special ontology of the design space and a generative grammar. (Special ontologies identify individuals, attributes, relationships, and classes defining relevant concepts of interest that establish a framework for reasoning within a specific domain.) As a special ontology, the language paradigm defines the structure and behavior that can be expressed as code (usually text). As a generative grammar the language paradigm defines the various grammatically equivalent expressions. Since code refactoring applies only meaning-invariant transformations, only transformations within that paradigm's special ontology that produce identical behavior (in terms of deterministic interpretation) are acceptable refactorings. Reframed by this unpacking, the definition of code refactoring is as follows:

> ***Code refactoring*** *applies design actions expressible in the ontological scope of the language paradigm to rectify deficits of design quality adherent to structural choice properties while preserving the code's behavior.*

As depicted in Figure 2, code refactoring applies to an extant artifact under the presumption that it faithfully represents stakeholders' intensions (i.e., it behaves appropriately). Refactoring is applied when structural aspects of the artifact are flawed; that when recognized as choice property deficiencies, they are remediated by design actions proposed to strengthen those deficient properties. In its purest use, code refactoring does not revisit or consider modification of the antecedent stakeholder intensions.

In the case of code refactoring Fowler, Beck, Kerievsky and others are sensitized to bad smells in artifacts rendered in the programming paradigm of object orientation. That paradigm adheres to a special ontology of object orientation as shown in Figure 3 [41, 42].

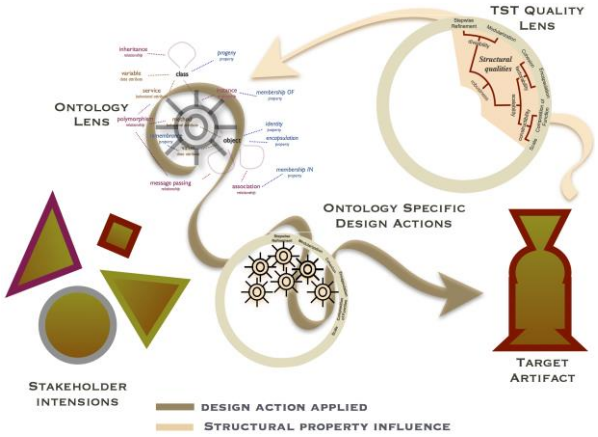

Figure 2 – Code Refactoring's Design Cycle

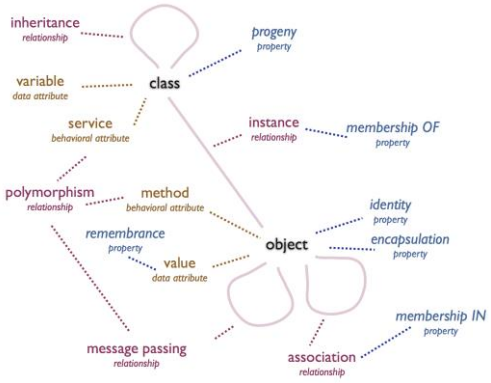

Figure 3 – Code Refactoring Rooted in the Ontology of Object Orientation

This object oriented (OO) ontology is the ontology lens depicted in Figure 2 that engenders the design actions pertinent in refactoring object oriented artifacts.

4.1 Refactoring as special purpose design

At its core, code refactoring is *redesign* restricted exclusively to those aspects of design choice properties that affect the facility of comprehending and manipulating working program code [14]. Dedicated as it is to preserving the behavior or functionality already coded, code refactoring is a narrowly focused demonstration of design. It is nonetheless a true exemplar of design by exhibiting:

(a) A purposeful succession of design actions (i.e. source code manipulations),
(b) Applied to form [or reform] an artifact (i.e. program code),
(c) Guided by a relevant value proposition (i.e. a theory of structural design quality).

In a given design space described by a special ontology: (i) the special ontology determines the available design actions and (ii) the theory of design quality interpreted with situational awareness directs the choices of design action; and together (i) and (ii) prescribe the nature and quality of the desired artifact.

A tacit sensitivity to design flaws and facility for discerning quality is clearly founded on the skill to converse in the domain of language. By whichever label (domain of language or language paradigm) the designer's ontological understanding of the medium of expression shapes her thinking, descriptions and conversations about the artifact in the design space – as marble to the sculptor, pastels to the painter, or musical notes to the composer.

Figure 2 depicts a design process consistent with Schön's [34] epistemology of *reflective practice*.

> "Three dimensions of this process are particularly noteworthy: the domains of language in which the designer describes and appreciates the consequences of his moves, the implications he discovers and follows, and his changing stance toward the situation with which he converses." [34, p. 95]

The role of a special ontology of construction in expressing artifact features influenced the rapid growth of object orientation as the shared special ontology across the software development life cycle. The consistent use of the OO ontology for the expression of requirements, design decisions, artifact testing, and artifact evolution helps programmers in what Naur calls *theory building*. On this journey, programmers "form or achieve a certain kind of insight, a theory, of the matters at hand." [26, p. 253]

In the refactoring literature [14, 20, 37], the programming paradigm of the code defines: (a) the domain of language and (b) the possible moves while (c) the skill to discern consequences and implications has developed over their many years of using the OO language paradigm. The changing stance is the perceived differential – *before* and *after* applying a refactoring. The combination of (a) and (b) serve de facto as their special ontology while (c) a theory of design quality for them (their appreciative system [10, p. A50]) has evolved tacitly.

4.2 Design refactoring in a cycle of reflective practice

Not all design quality deficits reside in the code itself. Some deficits originate in the antecedent stakeholder intensions that the code proposes to constitute or in the misalignment of the software's behavior with those intensions. Remediating intensional deficiencies must attend to more than resituating or reorienting syntactic elements. The design actions of the four refactorings (4 of the 80) that did not adhere to structural choice properties in Table 1 focused on clarifying the purpose of program code elements (renaming methods, adding assertions, and using names for numeric constants). These defects can neither be detected nor treated without revisiting the stakeholder intensions, or at least clarifying the understanding that arcs between descriptions of intension and the design goals targeted in the development, delivery, and acceptance of the artifact. These four instances of quality deficit adhere to the aesthetic choice property of *identity*. Where refactorings adhering to structural choice properties might be applied by code editors or IDE's performing "automatic" grammatical manipulations, these four can only be adjudged and treated through the stakeholders' aesthetic (subjective) sensibility, or the "capturing" thereof when design goals were elicited and articulated. While it may be true that "all the [structural] meaning resides in the text," the aesthetic choice properties adhere to a design space encompassing intension that can only be interpreted effectively through the stakeholders' experience of "what the text is intended to mean." Attending to such an expanded design space embraces a dramatically richer perspective on design quality (e.g. intension integrates meaning and purpose, purpose denotes value).

In our analysis of code refactoring and artifact quality the use of the term "stakeholder" naturally adheres to the designer or the author of the code. But the majority of interested parties making up the stakeholder community have no direct interest in the code. Their sense of quality is born primarily out of their individual experience with the design as materialized in the artifact.

This more experiential conception of design quality is what Alexander first called the "quality without a name," "QWAN," [1, p. 19] eventually addressing it as "wholeness." [2, p. 80] Refactoring an artifact as a whole (inclusive of intension and result) subsumes the stakeholders' preconception and perception of an artifact and reveals the breadth of their discernment of quality, their appreciative system [10]. Expanding refactoring beyond the "source code" to the full range of the stakeholders' experience of artifact quality requires design actions that strengthen choice properties to address not only the structural choice properties but, all

fifteen TST choice properties to enfold a comprehensive perspective of the stakeholders' disposition that shaped and shapes their intensions for the artifact, in other words, *the wholeness of the artifact*.

As depicted in Figure 4, after an initial artifact design, refactoring the wholeness proceeds by reflecting upon the artifact's qualities not only regarding structural properties, but also the artifact's resonance with the stakeholders' intensions. This reflection on wholeness detects flaws nominally associated with code refactoring, but also detects deficiencies that are misalignments between the artifact as realized and the strength of aesthetic choice properties the stakeholders expect. As Figure 4 indicates, the response to structure-based flaws continues to seek out design actions that address only the artifact as implemented. Enfolding aesthetic choice properties enables and insists that the reflective cycle contemplates the prospect that the antecedent stakeholder intensions may be constituent to a sense of a flawed design. Thus, the consequent refactoring must consider adjusting the expression of or essence of the intensions to resolve all choice property weaknesses or imbalances through design actions targeting the overall value of the artifact. Resolving aesthetically deficient stakeholder experiences routinely involves modifying artifact behavior. Furthermore, stakeholder intensions will likely evolve even prior to the initial fabrication of the target artifact as the stakeholder(s) and designer refine their intensions through cycles of reflection.

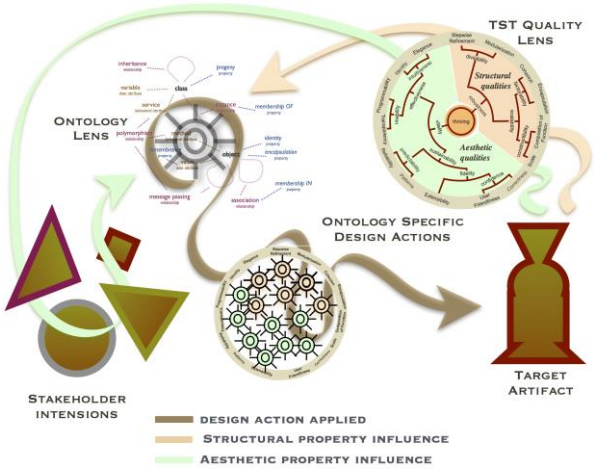

Figure 4 –Refactoring the Wholeness in a Reflective Practice Design Cycle

Each cycle of assessing the artifact's design quality, revisiting the antecedent intensions and applying design actions not only brings about the artifact's evolving transformation but also, continuously informs the designer's sense of the artifact's target quality as a resonance with the stakeholders' intensions. The reflective cycle informs both the ability to detect and diagnose deficiencies in efficacy and efficiency and to anticipate remediating design actions – what Schön refers to as the designer's *repertoire* [34]. This is the natural outcome of the designer *indwelling* in the design space of the artifact, seeking out the stakeholders' intensions as a whole and nurturing both a tacit and explicit capacity for skillfully aligning the artifact's properties with the stakeholder intensions (and vice versa). This design space reflects Alexander's conception of *wholeness.* It is this *wholeness* that is the objective of *design-as-a-verb.* [2, p. 80]

Such a broad notion of design as a reflective cycle represents a significantly more comprehensive process than code refactoring alone because of the inevitable evolution of intensions due to the virtually continuous changes in the environment of both the stakeholders and artifact.

> ***Design refactoring*** *applies design actions expressible in the ontological scope of the language paradigm to either of or both the artifact as rendered and the stakeholders' intensions to rectify deficits of design quality adherent to structural, behavioral, and aesthetic design concerns.*

Within Hevner's three-cycle model [17], design refactoring has the potential to be an important activity within the so-called *design cycle*, informing and being informed by the *relevance cycle*, within which most stakeholders determine an artifact's value. The special ontology shown in Figure 4 determines "what" range of possible artifacts may exist [16] while the theory of design quality explains "why" the stakeholders value the artifact.

Schön refers to the communication framed by the special ontology as a *reflective conversation*. In the case of a particular design project the vernacular derives from the language paradigm selected for describing or rendering the artifact and the designer's knowledge of the design space in question.

> "Media cannot really be separated in their influence from language and repertoire. Together they make up the "stuff" of inquiry, in terms of which practitioners move, experiment, and explore. Skills in the manipulation of media, languages and repertoires are essential to a practitioner's reflective conversation with her situation, just as skill in the manipulation of spoken language is essential to ordinary conversation. […] Because they have developed a feel for the media and languages of their practices, the individuals we have studied can construct virtual worlds in which to carry out imaginative rehearsals of action." [34, p. 271]

5. Shaping design as reflective conversation

As we conclude our exercise decomposing code refactoring to extract its founding principles and operative behaviors, we are drawn to contemplate what activity in practice the *reflective conversation refactoring the design* would entail. We choose the terms "conversation," and "recipe" in a concerted attempt to avoid any appearance of prescription. Both terms connote a sense of "becoming" rather than consummation. In that sense we focus on design as a "living cycle" rather than a product of construction. At its core, the "becoming" conversation echoes the Scrum premise that refining the goals of the development project depends upon a conversation between product owner, scrum master, and development team as the evolving artifact is repeatedly assessed and the product backlog and sprint planning react to the progressive learning and refinement of the project goals among the stakeholders [32].

The three essential ingredients in a recipe of design as conversation are: a communicable depiction of the stakeholder intensions – an *expression of intent*, an ontological depiction of the design space consistent with that expression, and an applicable appreciative system (theory of design quality) with which to guide the selection of design actions reflective of quality design choices. The recipe that follows is a sketch of *refactoring design as a reflective conversation*.

Reflection-Driven Design Recipe

(A) Form an expression of intent describing the desired structure and behavior of the artifact.
(B) Specify an ontological depiction of the design space consistent with the expression of intent.
(C) Define a collection of design choice properties representing the stakeholders' collective appreciative system with which to assess quality.
(D) Interpret the choice properties of the appreciative system through the design actions determined by the ontological depiction.
(E) Construct [or modify] the artifact by applying design actions that evince the desired choice properties.
(F) Reflect upon the structural and behavioral quality of the artifact in terms of the choice properties.
(G) While the artifact's structure and behavior do not faithfully represent the expression of intent iterate through (D)-(E)-(F)-(G).
(H) If the artifact's behavior is faithful to the expression of intent but not "satisfactory" then restart the design cycle from (A).
(I) Resume at (F) as needed for the "life" of the artifact.

The following elaborates further upon each of the nine activities by referencing the keyed elements of the reflective conversation in Figure 5.

(A) Design is the act of conceiving the existence of an artifact that reflects an idea held by one or more stakeholders. The idea (particularly if shared among more than one person) must be expressed in a form explicitly providing shared access and the opportunity to seek consensus. In as much as the builders may be separate from the source of the ideas, the expression is the postulated reality that the artifact is to become.

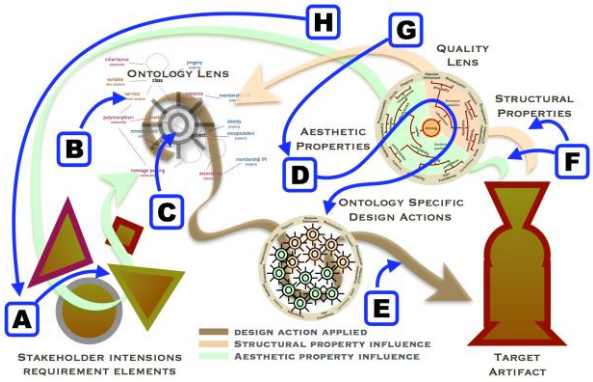

Figure 5 – Reflective Conversation Unpacked

(B) As the expression of intent may be of any particular dialect or paradigm, the task of constructing the artifact requires defining the characteristics of its construction: the elements that will form its structure and support the behavior that will determine the experience that users and stakeholders will deem acceptable and/or satisfying.

(C) The nature of satisfaction lies in the individual stakeholder's appreciative system, those values that are held as necessary or desirable. Values are the product of culture and lived experience both individually and in community. Although in many instances individuals hold some values tacitly, those values that determine acceptability or satisfaction in community must become explicit to the extent that they are operable in an assessment of quality. Choice properties represent the dimensions of features or characteristics of an artifact that reflect the appreciative system and must be localized to the design activity at hand.

(D) Where choice properties reflect dimensions of value, design must contemplate actions that impact the structure and behavior of the artifact. Design actions are rooted in a special ontology of construction delineating resources and constructive actions specific to the design space. Designers need to know which and how to apply design actions in the "world" the artifact and stakeholders will share in order to strengthen (or

weaken) the choice properties that reflect their values. Without such a correlation between design actions and choice properties, design would simply be undirected trial and error.

(E) Applying design actions is the constructive act of shaping the features and characteristics that advance the artifact in its "becoming." With each action applied the artifact advances or retreats from an alignment with the expression of intent and the degree of satisfaction manifest in its current form.

(F) Reflection is the deliberate analysis of the artifact's degree of alignment with the expression of intent. It is guided by the choice properties that reflect the appreciative system that mediates the stakeholder community's value system.

(G) If the reflection reveals a degree of alignment between artifact and expression of intent that is deficient the conversation returns to examining the relevant choice properties and then the design actions that may be applied to remediate that deficit. In effect the design cycle reverts to (D)-(E)-(F)-(G).

(H) If the reflection on the artifact and expression of intent reveals they are effectively aligned, yet the degree of satisfaction experienced is in deficit, reflection must turn to the conceptual premise of the design effort. Some aspect of the expression of intent, the ontological expression of the design space, or the interpretation of the choice properties through design actions, must be conceptualized anew. The design effort must return to (A) and reconsider the artifact's design premise.

(I) At this point in the design recipe the artifact reflects "satisfaction" and the current design is considered successful. But as satisfaction is a human experience in a dynamic world, the alignment of the artifact with the stakeholders' intensions inevitably erodes as surely as entropy increases according to the 2nd law of thermodynamics. As long as the artifact is relevant to the stakeholder community, its value deserves review and reconsideration. This is a commitment to refreshing the artifact's relevance and satisfaction – treating the artifact as a living asset.

5.1 Considering the design recipe in a context

With the above explication of the design recipe, we provide some further elaboration using some notional examples from Agile development process models and methods such as XP [6] and Scrum [32]. Utilizing the steps of the recipe, and their orientation expressed in Figure 5. Here are some further ruminations offered for illustration.

Recipe activity (A): In a contemporary context, as would be the case with an Agile software development process such as Scrum, artifact elicitation and realization is collective between a team of developers, intercessors (e.g., product owners and scrum masters), and other stakeholders. Through this early and close contact, developers, by virtue of exposure and focus, will develop tacit familiarity with a broad range of concerns that may warrant subsequent refactoring: concerns that might be structural, behavioral, or aesthetic in nature. As user stories are developed, prioritized, assessed and ordered – even if within the product owner's purview – the team develops early awareness of the product's context, which is tethered to the ontological lens and the set of actions implied by and required by this lens. Thus, there is immediately a juxtaposition between past experience – manifest in the team members' repertoire – and the elicitation of current stakeholder intensions, that will inform reflective judgment and deterministic judgment [24], and perhaps expose technical debt [8, 37].

Recipe activity (B)*:* To continue to use Agile development process models for illustration, XP particularly recognizes the need for this activity in our recipe in its traditions and concepts of architectural spike and system metaphor [6]. Both are introduced as means of developing team orientation towards an iterative realization of the product at hand through testing and confirming against prior knowledge to determine novel items and distinguish them from known patterns and techniques for success. There is little wonder then that the refactoring phenomenon emphasizes refactoring to patterns [20] as this too reflects an espoused desire to "tie" the project "down" to minimize risk and uncertainty.

Recipe activity (C): There is a theme which pervades many Agile methods that espouses team empowerment. Even as Scrum would depend on the product owner – a voice of the customer and also funnel and filter for intensions – the conveyance of the values underpinning satisfaction will always be indirect and translated [32]. As an appreciative system operates dynamically; understanding must be recreated on an ongoing basis [10].

Recipe activities (D) *and* (E): Within Scrum, the emergence through iterating in a sprint, there are prescriptions for daily practice to check on the correlation, and alignment, of design actions and choice properties. Design, construction, testing, integration, and reflective consideration are important components of a continuous delivery cycle that has come to be as Agile development has evolved to promulgate the DevOps movement [21].

Recipe activities (F), (G), *and* (H): Agile methods also espouse adherence to an epistemology of reflective practice as is evidenced in the 12th principle in the original Agile Manifesto [7]. By their very nature, the processes of XP [6] and Scrum [32] require pauses for inspection and assessment to ensure harmony and

alignment. They also allow for changes in direction, velocity, and extent that arise from these activities. Moreover, in consideration of the evolution of the artifact, the stakeholders' appreciative system would be prone to exerting change on the artifact as the artifact has likely influenced and changed facts surrounding the world-view of the stakeholders.

Recipe activity (I): While many Agile methods assume that the method is to be employed to move a software or system project through to completion, as an evolution of Agile methods in their application, DevOps has arisen to recognize a degree of perpetuity in how the team (designers & developers) becomes attuned to systems thinking, sensitivity to feedback, and ensuing continuous learning [21]. The difference between code refactoring and design refactoring is that code refactoring stipulates the immutability of the artifact's behavior and therefore never questions the expression of intent. The purpose of code refactoring is to perfect the implementation of the initial expression of intent which reflects Argyris and Schön's conception of *single loop learning* [4]. When a reflective conversation reverts to reflecting anew upon the expression of intent as the possible cause of satisfaction or quality deficit and the potential of modifying both the structural and behavioral premises expressed – that reflects *double loop learning* [4, 31].

While all artifacts may eventually cease to exist, their lifetime likely extends beyond the project as a scope of concern. This longitudinal aspect further suggests that, whether within the tacit knowledge of the individual developer, or manifest within the wider lore of the team, a collection and culmination of perspectives, proclivities, and propensities regarding the full spectrum of design (and design refactoring) will likely persist. It is difficult to conceive that code refactoring, even if it indwells principally within structural aspects of a design, will not spill over into a broader spectrum of design concerns. Since Agile methods are particularly "tuned in" to this possibility, we believe there are broader opportunities to "refactor" our own discourse regarding the role code refactoring plays in the overall design of systems. Specifically, we advocate for an empowered developer who, in her reflection-in-action and -on-action, often engages in double-loop learning to refactor stakeholder intensions, and subsequently engage stakeholders in iterative design [4, 6, 8, 15, 17, 22, 24, 31, 34].

6. Discussion and conclusion

Figures 2 and 3 above reflect the special ontology of object orientation because that is the design space of the artifact addressed in code refactoring. If, as we propose, design is best conceived of as a *reflective conversation refactoring the wholeness of an artifact* (see Figures 4 and 5), then the special ontology that is appropriate is dictated by the design space of the artifact. Such special ontologies for design propose design action guidelines aligned to the fifteen TST choice properties in Table 1. Our ongoing research applies domain specific ontologies to entity relationship modeling and information systems security to extend *design refactoring* into those domains [42].

In this paper, we explored the design quality relating code refactorings [14, 20, 37] and the choice properties of TST [40]. We examined refactoring as design actions prescribed by the special ontology of the design space. Such an ontology determines "what" range of possible artifacts may exist while the theory of design quality (e.g., TST) explains "why" the stakeholders value the artifact. Nearly all of the 80 proffered refactorings defined in [14, 20] understandably adhere to the six choice properties that relate to the structural concerns of design quality. A human perception of the wholeness of an artifact's design quality, however, extends beyond the structural [2, 40] to an appreciative system [10]. By examining code refactoring using the 15 choice properties of TST, we were able to generalize code refactoring to design refactoring, conceptualizing design and design quality to a wider range of concerns.

We describe an iterative process of design as a cycle of reflective practice that integrates structural, behavioral, and aesthetic aspects of design quality as a natural evolution of code refactoring. In software development and delivery, this iterative process creates positive feedback between programming and theory building [26]. Theory is built and developed as part of a programmer's, her team's, and her community's knowledge [17, 21]. We posit that the formula demonstrated by Fowler, Beck, Kerievsky and others for identifying technical debt in the form of bad smells, or even design debt, and consequent prescriptions for remedial design actions, can be replicated by extending the formula to include a broader spectrum of design concerns. Doing so provides a more comprehensive foundation for both effective design practice and also design pedagogy.

7. References

[1] Alexander, C. W. (1979). *A Timeless Way of Building*. Oxford University Press, New York, NY.

[2] Alexander, C. W. (2002). *The Nature of Order An Essay on the Art of Building and the Nature of the Universe: Book I – The Phenomenon of Life*. The Center for Environmental Structure, Berkley, CA.

[3] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., & Lucena, C. (2006). "Refactoring Product Lines." In Proc. of the *5th International Conference on Generative Programming and Component Engineering*, pp. 201-210.

[4] Argyris, C., & Schön, D. (1978). *Organizational Learning: A Theory of Action Perspective*. McGraw-Hill, New York, NY.
[5] Baldwin, C. Y., & Clark, K. B. (2000). *Design Rules, Volume 1: The Power of Modularity*. MIT Press, Cambridge, MA.
[6] Beck, K. (2004). *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, MA.
[7] Beck, K. et al. (2001). *Manifesto for Agile Software Development*. Agile Alliance. Retrieved 3 January 2016.
[8] Brown, N., Cai, Y., Guo, Y., Kazman, R. et al. (2010). "Managing Technical Debt in Software-Reliant Systems." In Proc. of the *FSE/SDP Workshop on Future of Software Engineering Research*, pp. 47-51.
[9] Chandrasekaran, B., Josephson, J. R., & Benjamins, V. R. (1999). "What Are Ontologies, and Why Do We Need Them?" *IEEE Intelligent Systems*, 14(1), pp. 20-26.
[10] Checkland, P. (1999). *Systems Thinking, Systems Practice*. John Wiley & Sons, New York, NY.
[11] Dennis, A., Wixom, B. H., & Roth, R. M. (2014). *Systems Analysis and Design*, 6th ed. John Wiley & Sons, New York, NY.
[12] Dingsøyr, T., Nerur, S., Balijepally, V., & Moe, N. B. (2012). "A Decade of Agile Methodologies: Towards Explaining Agile Software Development," *Journal of Systems and Software*, 85(6), pp. 1213-1221.
[13] Elish, K. O., & Alshayeb, M. (2011). "A Classification of Refactoring Methods Based on Software Quality Attributes," *Arab Journal of Science and Engineering*, 36, pp. 1253-1267.
[14] Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA.
[15] Gilb, T., & Finzi, S. (1988). *Principles of Software Engineering Management*. Addison-Wesley, Reading, MA.
[16] Gregor, S., & Jones, D. (2007). "The Anatomy of a Design Theory," *Journal of the Association for Information Systems*, 8(5), pp. 312-335.
[17] Hevner, A. R. (2007). "A Three Cycle View of Design Science Research," *Scandinavian Journal of Information Systems*, 19(2), pp. 87-92.
[18] Janzen, D., & Saiedian, H. (2005). "Test-Driven Development: Concepts, Taxonomy, and Future Direction," *Computer*, 38(9), pp. 43-50.
[19] Jensen, A. C., & Cheng, Betty H. C. (2010). "On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns." In Proc. of the *12th Annual Conference on Genetic and Evolutionary Computation*, pp. 1341-1348.
[20] Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley, Reading, MA.
[21] Kim, G., Debois, P., Willis, J., & Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, Portland, OR.
[22] Larman, C., & Basili, V. R. (2003). "Iterative and Incremental Development: A Brief History," *Computer*, 36(6), pp. 47-56.
[23] Lee, A. S., & Baskerville, R. L. (2003). "Generalizing Generalizability in Information Systems Research," *Information Systems Research*, 14(3), pp. 221-243.
[24] Lee, J. S., Pries-Heje, J., & Baskerville, R. (2011). "Theorizing in Design Science Research," In *Service-Oriented Perspectives in Design Science Research*, Lecture Notes in Computer Science, 6629, pp. 1-16.
[25] Mens, T., & Tourwé, T. (2004). "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, 30(2), pp. 126-139.
[26] Naur, P. (1985). "Programming as Theory Building," *Microprocessing and Microprogramming*, 15(5), pp. 253–261.
[27] Nerur, S., & Balijepally, V. (2007). "Theoretical Reflections on Agile Development Methodologies," *Communications of the ACM*, 50(3), pp. 79-83.
[28] Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks* (Doctoral dissertation, University of Illinois at Urbana-Champaign).
[29] Pham, H. (2000). *Software Reliability*. Springer, Berlin, Germany.
[30] Polanyi, M. (1966). *The Tacit Dimension*. University of Chicago Press, Chicago, IL.
[31] Ramasubbu, N., Mithas, S., Krishnan, M. S., & Kemerer, C. F. (2008). "Work Dispersion, Process-Based Learning, and Offshore Software Development Performance," *MIS Quarterly*, 32(2), pp. 437-458.
[32] Rubin, K. S. (2012). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley, Reading, MA.
[33] Schmidt, D. C. (2006). "Model-driven engineering," *Computer*, 39(2), pp. 25-31.
[34] Schön, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, New York, NY.
[35] Sullivan, K. J., Griswold, W. G., Cai, Y., & Hallen, B. (2001). "The Structure and Value of Modularity in Software Design," *ACM SIGSOFT Software Engineering Notes*, 26(5), pp. 99-108.
[36] Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.-M. (2001). "Refactoring UML Models." In Proc. of *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pp. 134-148.
[37] Suryanarayana, G., Samarthyam G., & Sharma T. (2014). *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, New York, NY.
[38] Uschold, M., & Gruninger, M. (1996). "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review*, 11(2), pp. 93-136.
[39] Verganti, R. (2009). *Design-Driven Innovation: Changing The Rules of Competition by Radically Innovating What Things Mean*. Harvard Business Press, Cambridge, MA.
[40] Waguespack, L. J. (2010). *Thriving Systems Theory and Metaphor-Driven Modeling*. Springer-Verlag, London, UK. (ISBN: 978-1-84996-301-5)
[41] Waguespack, L. J. (2015). "A Design Quality Learning Unit in OO Modeling Bridging the Engineer and the Artist," *Information Systems Education Journal*, 13(1), pp. 58-70.
[42] Waguespack, L. J. (2016). "IS Design Pedagogy: A Special Ontology and Prospects for Curricula," *Information Systems Education Journal*, 14(6), pp. 4-13.
[43] Winograd, T., & Flores, F. (1986). *Understanding Computers and Cognition: A New Foundation for Design*. Addison-Wesley, Reading, MA.
[44] Zuse, H. (1997). *A Framework of Software Measurement*. Walter de Gruyter, Berlin, Germany.